**Top Ten Errors Java Programmers Make**

### 10. Accessing non-static member variables from static methods (such as main)

Many programmers, particularly when first introduced to Java, have problems with accessing member variables from their main method. The method signature for main is marked static - meaning that we don't need to create an instance of the class to invoke the main method. For example, a Java Virtual Machine (JVM) could call the class MyApplication like this :-

MyApplication.main ( command\_line\_args );

This means, however, that there isn't an instance of MyApplication - it doesn't have any member variables to access! Take for example the following application, which will generate a compiler error message.

public class StaticDemo

{

public String my\_member\_variable = "somedata";

public static void main (String args[])

{

// Access a non-static member from static method

System.out.println ("This generates a compiler error" +

my\_member\_variable );

}

}

If you want to access its member variables from a non-static method (like main), you must create an instance of the object. Here's a simple example of how to correctly write code to access non-static member variables, by first creating an instance of the object.

public class NonStaticDemo

{

public String my\_member\_variable = "somedata";

public static void main (String args[])

{

NonStaticDemo demo = new NonStaticDemo();

// Access member variable of demo

System.out.println ("This WON'T generate an error" +

demo.my\_member\_variable );

}

}

### 9. Mistyping the name of a method when overriding

Overriding allows programmers to replace a method's implementation with new code. Overriding is a handy feature, and most OO programmers make heavy use of it. If you use the AWT 1.1 event handling model, you'll often override listener implementations to provide custom functionality. One easy trap to fall into with overriding, is to mistype the method name. If you mistype the name, you're no longer overriding a method - you're creating an entirely new method, but with the same parameter and return type.

public class MyWindowListener extends WindowAdapter {

// This should be WindowClose**d**

public void WindowClose(WindowEvent e) {

// Exit when user closes window

System.exit(0);

}

});

Compilers won't pick up on this one, and the problem can be quite frustrating to detect. In the past, I've looked at a method, believed that it was being called, and taken ages to spot the problem. The symptom of this error will be that your code isn't being called, or you think the method has skipped over its code. The only way to ever be certain is to add a println statement, to record a message in a log file, or to use good trace debugger (like Visual J++ or Borland JBuilder) and step through line by line. If your method still isn't being called, then it's likely you've mistyped the name.

### 8. Comparison assignment (  = rather than == )

This is an easy error to make. If you're used other languages before, such as Pascal, you'll realize just how poor a choice this was by the language's designers. In Pascal, for example, we use the := operator for assignment, and leave = for comparison. This looks like a throwback to C/C++, from which Java draws its roots.

Fortunately, even if you don't spot this one by looking at code on the screen, your compiler will. Most commonly, it will report an error message like this : "Can't convert xxx to boolean", where xxx is a Java type that you're assigning instead of comparing.

### 7. Comparing two objects ( == instead of .equals)

When we use the == operator, we are actually comparing two object references, to see if they point to the same object. We cannot compare, for example, two strings for equality, using the == operator. We must instead use the .equals method, which is a method inherited by all classes from java.lang.Object.

Here's the correct way to compare two strings.

String abc = "abc"; String def = "def";

// Bad way

if ( (abc + def) == "abcdef" )

{

    ......

}

// Good way

if ( (abc + def).equals("abcdef") )

{

   .....

}

### 6. Confusion over passing by value, and passing by reference

This can be a frustrating problem to diagnose, because when you look at the code, you might be sure that its passing by reference, but find that its actually being passed by value. Java uses **both**, so you need to understand when you're passing by value, and when you're passing by reference.

When you pass a primitive data type, such as a char, int, float, or double, to a function then you are **passing by value**. That means that a copy of the data type is duplicated, and passed to the function. If the function chooses to modify that value, it will be modifying the copy only. Once the function finishes, and control is returned to the returning function, the "real" variable will be untouched, and no changes will have been saved. If you need to modify a primitive data type, make it a return value for a function, or wrap it inside an object.

When you pass a Java object, such as an array, a vector, or a string, to a function then you are **passing by reference**. Yes - a String is actually an object, not a primitive data type.  So that means that if you pass an object to a function, you are passing a reference to it, not a duplicate. Any changes you make to the object's member variables will be permanent - which can be either good or bad, depending on whether this was what you intended.

On a side note, since String contains no methods to modify its contents, you might as well be passing by value.

### 5. Writing blank exception handlers

I know it's very tempting to write blank exception handlers, and to just ignore errors. But if you run into problems, and haven't written any error messages, it becomes almost impossible to find out the cause of the error. Even the simplest exception handler can be of benefit. For example, put a try { .. } catch Exception around your code, to catch ANY type of exception, and print out the message. You don't need to write a custom handler for every exception (though this is still good programming practice). Don't ever leave it blank, or you won't know what's happening.

For example

public static void main(String args[])

{

    try {

// Your code goes here..

    }

    catch (Exception e)

    {

System.out.println ("Err - " + e );

    }

}

### 4. Forgetting that Java is zero-indexed

If you've come from a C/C++ background, you may not find this quite as much a problem as those who have used other languages. In Java, arrays are zero-indexed, meaning that the first element's index is actually 0. Confused? Let's look at a quick example.

// Create an array of three strings

String[] strArray = new String[3];

// First element's index is actually 0

strArray[0] = "First string";

// Second element's index is actually 1

strArray[1] = "Second string";

// Final element's index is actually 2

strArray[2] = "Third and final string";

In this example, we have an array of three strings, but to access elements of the array we actually subtract one. Now, if we were to try and access strArray[3], we'd be accessing the fourth element. This will case an ArrayOutOfBoundsException to be thrown - the most obvious sign of forgetting the zero-indexing rule.

Other areas where zero-indexing can get you into trouble is with strings. Suppose you wanted to get a character at a particular offset within a string. Using the String.charAt(int) function you can look this information up - but under Java, the String class is also zero-indexed. That means than the first character is at offset 0, and second at offset 1. You can run into some very frustrating problems unless you are aware of this - particularly if you write applications with heavy string processing. You can be working on the wrong character, and also throw exceptions at run-time. Just like the ArrayOutOfBoundsException, there is a string equivalent. Accessing beyond the bounds of a String will cause a StringIndexOutOfBoundsException to be thrown, as demonstrated by this example.

public class StrDemo

{

public static void main (String args[])

{

String abc = "abc";

System.out.println ("Char at offset 0 : " + abc.charAt(0) );

System.out.println ("Char at offset 1 : " + abc.charAt(1) );

System.out.println ("Char at offset 2 : " + abc.charAt(2) );

// This line should throw a StringIndexOutOfBoundsException

System.out.println ("Char at offset 3 : " + abc.charAt(3) );

}

}

Note too, that zero-indexing doesn't just apply to arrays, or to Strings. Other parts of Java are also indexed, but not always consistently. The java.util.Date, and java.util.Calendar classes start their months with 0, but days start normally with 1. This problem is demonstrated by the following application.

import java.util.Date;

import java.util.Calendar;

public class ZeroIndexedDate

{

public static void main (String args[])

{

// Get today's date

Date today = new Date();

// Print return value of getMonth

System.out.println ("Date.getMonth() returns : " +

today.getMonth());

// Get today's date using a Calendar

Calendar rightNow = Calendar.getInstance();

// Print return value of get ( Calendar.MONTH )

System.out.println ("Calendar.get (month) returns : " +

rightNow.get ( Calendar.MONTH ));

}

}

Zero-indexing is only a problem if you don't realize that its occurring. If you think you're running into a problem, always consult your API documentation.

### 3. Preventing concurrent access to shared variables by threads

When writing multi-threaded applications, many programmers (myself included) often cut corners, and leave their applications and applets vulnerable to thread conflicts. When two or more threads access the same data concurrently, there exists the possibility (and Murphy's law holding, the probability) that two threads will access or modify the same data at the same time. Don't be fooled into thinking that such problems won't occur on single-threaded processors. While accessing some data (performing a read), your thread may be suspended, and another thread scheduled. It writes its data, which is then overwritten when the first thread makes its changes.

Such problems are not just limited to multi-threaded applications or applets. If you write Java APIs, or JavaBeans, then your code may not be thread-safe. Even if you never write a single application that uses threads, people that use your code WILL. For the sanity of others, if not yourself, you should always take precautions to prevent concurrent access to shared data.

How can this problem be solved? The simplest method is to make your variables private (but you do that already,  right?) and to use synchronized accessor methods. Accessor methods allow access to private member variables, but in a controlled manner. Take the following accessor methods, which provide a safe way to change the value of a counter.

public class MyCounter

{

private int count = 0; // count starts at zero

public synchronized void setCount(int amount)

{

count = amount;

}

public synchronized int getCount()

{

return count;

}

}

### 2. Capitalization errors

This is one of the most frequent errors that we all make. It's so simple to do, and sometimes one can look at an uncapitalized variable or method and still not spot the problem. I myself have often been puzzled by these errors, because I recognize that the method or variable does exist, but don't spot the lack of capitalization.

While there's no silver bullet for detecting this error, you can easily train yourself to make less of them. There's a very simple trick you can learn :-

* all methods and member variables in the Java API begin with lowercase letters
* all methods and member variables use capitalization where a new word begins e.g - getDoubleValue()

If you use this pattern for all of your member variables and classes, and then make a conscious effort to get it right, you can gradually reduce the number of mistakes you'll make. It may take a while, but it can save some serious head scratching in the future.

## (drum roll)

## And the number one error that Java programmers make !!!!!

**1. Null pointers!**

Null pointers are one of the most common errors that Java programmers make. Compilers can't check this one for you - it will only surface at runtime, and if you don't discover it, your users certainly will.

When an attempt to access an object is made, and the reference to that object is null, a NullPointerException will be thrown. The cause of null pointers can be varied, but generally it means that either you haven't initialized an object, or you haven't checked the return value of a function.

Many functions return null to indicate an error condition - but unless you check your return values, you'll never know what's happening. Since the cause is an error condition, normal testing may not pick it up - which means that your users will end up discovering the problem for you. If the API function indicates that null may be returned, be sure to check this before using the object reference!

Another cause is where your initialization has been sloppy, or where it is conditional. For example, examine the following code, and see if you can spot the problem.

public static void main(String args[])

{

// Accept up to 3 parameters

String[] list = new String[3];

int index = 0;

while ( (index < args.length) && ( index < 3 ) )

{

list[index++] = args[index];

}

// Check all the parameters

for (int i = 0; i < list.length; i++)

{

if (list[i].equals "-help")

{

// .........

}

else

if (list[i].equals "-cp")

{

// .........

}

// else .....

}

}

This code (while a contrived example), shows a common mistake. Under some circumstances, where the user enters three or more parameters, the code will run fine. If no parameters are entered, you'll get a NullPointerException at runtime. Sometimes your variables (the array of strings) will be initialized, and other times they won't. One easy solution is to check BEFORE you attempt to access a variable in an array that it is not equal to null.

### Summary

These errors represent but some of the many that we all make. Though it is impossible to completely eliminate errors from the coding process, with care and practice you can avoid repeating the same ones. Rest assured, however, that all Java programmers encounter the same sorts of problems. It's comforting to know, that while you work late into the night tracking down an error, someone, somewhere, sometime, will make the same mistake!

|  |
| --- |
| We'd like to thank the readers of the comp.lang.java.programmer newsgroup for their suggestions for the top ten. Regrettably, due to the number of submissions, not every error could be featured - but we think this "Top Ten" list represents the most popular and frequent errors people make. |